Chapter 3

Greedy algorithms

This chapter explains the reasoning in finding optimal greedy algorithms. The main feature of a greedy algorithm is that it builds the solution step by step, and, at each step, it makes a decision that is locally optimal. Throughout Sections 3.1 to 3.3, we illustrate this principle with several examples, and also outline situations where greedy algorithms are not optimal; taking a good local decision may prove a bad choice in the end! In Section 3.4, we also cover matroids, a (mostly theoretical) framework to prove the optimality of greedy algorithms. All of these techniques are then illustrated with a set of exercises in Section 3.5, with solutions found in Section 3.6.

3.1 Motivating example: The sports hall

Problem. Let us consider a sports hall in which several events should be scheduled. The goal is to have as many events as possible, given that two events cannot occur simultaneously (only one hall). Each event $i$ is characterized by its starting time $s_i$ and its ending time $e_i$. Two events are compatible if their time intervals do not overlap. We would like to solve the problem, i.e., find the maximum number of events that can fit in the sports hall, with a greedy algorithm.

A first greedy algorithm. The first idea consists in sorting events by increasing durations $e_i - d_i$. At each step, we schedule an event into the sports hall if it fits, i.e., if it is compatible with events that have already been scheduled. The idea is that we will be able to accommodate more shorter events than longer ones. However, we make local decisions at each step of the algorithm (this is a greedy algorithm!), and it turns out that we can make decisions that do not lead to the optimal solution. For instance, in the example of Figure 3.1, the greedy algorithm schedules only the shortest event $i$, while the two compatible events $j$ and $k$ would lead to a better solution.

A second greedy algorithm. In order to avoid the problem encountered in the previous example, we design a new algorithm that sorts events by starting
Chapter 3. Greedy algorithms

The first greedy algorithm is not optimal. Times $s_i$, and then proceeds similarly to the first greedy algorithm. In the example of Figure 3.1, this greedy algorithm returns the optimal solution. However, the local decisions that are made may not be the optimal ones, as shown in the example of Figure 3.2. Indeed, the algorithm schedules event $i$ at the first step, and then no other event can be scheduled, while it would be possible to have eight compatible events. Note that the first greedy algorithm would return the optimal solution for this example.

A third greedy algorithm. Building upon the first two algorithms, we observe that it is always a good idea to first select events that do not intersect with many other events. In the first example, events $j$ and $k$ intersect with only one other event, while event $i$ intersects with two events and is chosen later; therefore, the new algorithm finds the optimal solution. Similarly in the second example, event $i$ intersects eight other events and it is the only event not to be scheduled. However, this greedy algorithm is still not optimal. We can build an example in which we force the algorithm to make a bad local decision. In the example of Figure 3.3, event $i$ is the first to be chosen because it has the smallest number of intersecting events. However, if we schedule $i$, we can have only three compatible events, while we could have a solution with four compatible events, $j$, $k$, $l$, and $m$.

FIGURE 3.1: The first greedy algorithm is not optimal.

FIGURE 3.2: The second greedy algorithm is not optimal.

FIGURE 3.3: The third greedy algorithm is not optimal.
3.2 Designing greedy algorithms

An optimal greedy algorithm. Even though many greedy choices do not lead to an optimal solution, as observed with the preceding algorithms, there is a greedy algorithm that solves the sports hall problem in polynomial time. The idea is to sort the events by increasing ending times $e_i$, and then to greedily schedule the events. This way, at each step we fit the maximum number of events up to a given time, and we never make a bad decision. We now prove the optimality of this algorithm.

Let $f_1$ be the event with the smallest ending time. We prove first that there exists an optimal solution that schedules this event. Let us consider an optimal solution $O = \{ f_{i_1}, f_{i_2}, \ldots, f_{i_k} \}$, where $k$ is the maximum number of events that can be scheduled in the sports hall, and where events are sorted by nondecreasing ending times. There are two possible cases: either (i) $f_{i_1} = f_1$, the optimal solution schedules $f_1$, and nothing needs to be done, or (ii) $f_{i_1} \neq f_1$. In this second case, we replace $f_{i_1}$ with $f_1$ in solution $O$. We have $e_1 \leq e_{i_1}$ by definition of event $f_1$, and $e_{i_1} \leq s_{i_2}$ because $O$ is a solution to the problem ($f_{i_1}$ and $f_{i_2}$ are compatible). Therefore, $e_1 \leq s_{i_2}$ and, thus, $f_{i_2}$ is compatible with $f_1$. The new solution is still optimal (the number of events remain unchanged), and event $f_1$ is scheduled.

The proof works by induction, following the previous reasoning. Once $f_1$ is scheduled, we consider only events that do not intersect with $f_1$, and we iterate the reasoning on the remaining events to conclude the proof.

Finally, we emphasize that there can be many optimal solutions, and not all of them will include the first event $f_1$ selected by the greedy algorithm, namely the event with the smallest end time. However, schedules that select $f_1$ are dominant, meaning that there exists an optimal solution that includes $f_1$.

3.2 Designing greedy algorithms

The example of the sports hall gives a good introduction to the design principles of greedy algorithms. Actually, the binary method to compute $x^n$ in Section 1.1.2 also is a greedy algorithm, in which we decide at each step which computation to perform. We can formalize the reasoning to find greedy algorithms as follows:

1. Decide on a greedy choice that allows us to locally optimize the problem;
2. Search for a counterexample that shows that the algorithm is not optimal (and go back to step 1 if a counterexample is found), or prove its optimality through steps 3 and 4;
3. Show that there is always an optimal solution that performs the greedy choice of step 1;
4. Show that if we combine the greedy choice with an optimal solution of the subproblem that we still need to solve, then we obtain an optimal solution.

We say that a greedy algorithm is a top-down algorithm, because at each step we make a local choice, and we then have a single subproblem to solve, given this choice. On the contrary, we will see in Section 4 that dynamic programming algorithms are bottom-up; we will need results of multiple subproblems to make a choice.

3.3 Graph coloring

In this section, we further illustrate the principle of greedy algorithms through the example of graph coloring. The problem consists in coloring all vertices of a graph using the minimum number of colors, while enforcing that two vertices, which are connected with an edge, are not of the same color. Formally, let $G = (V,E)$ be a graph and $c : V \rightarrow \{1..K\}$ be a $K$-coloring such that $(x,y) \in E \Rightarrow c(x) \neq c(y)$. The objective is to minimize $K$, the number of colors.

3.3.1 On coloring bipartite graphs

We start with a small theorem that allows us to define a bipartite graph, defined as a graph that can be colored with only two colors.

**Theorem 3.1.** A graph can be colored with two colors if and only if all its cycles are of even length.

**Proof.** Let us first consider a graph $G$ that can be colored with two colors. Let $c(v) \in \{1,2\}$ be the color of vertex $v$. We prove by contradiction that all cycles are of even length. Indeed, if $G$ has a cycle of length $2k + 1$, $v_1, v_2, \ldots, v_{2k+1}$, then we have $c(v_1) = 1$, say, which implies that $c(v_2) = 2$, $c(v_3) = 1$, until $c(v_{2k+1}) = 1$. However, since it is a cycle, there is an edge between $v_1$ and $v_{2k+1}$, so they cannot be of the same color, which leads to the contradiction.

Now, if all the cycles of the graph $G$ are of even length, we search for a 2-coloring of this graph. We assume that $G$ is connected (the problem is independent from one connected component to another). The idea consists in performing a breadth-first traversal of $G$.

Let $x_0 \in G$, $X_0 = \{x_0\}$ and $X_{n+1} = \bigcup_{y \in X_n} N(y)$, where $N(y)$ is the set of nodes connected to $y$, but not yet included in a set $X_k$, for $k \leq n$. Each vertex appears in one single set, and we color with color 1 the elements from sets $X_{2k}$, and with color 2 the elements from sets $X_{2k+1}$.
3.3. Graph coloring

This 2-coloring is valid if and only if two vertices connected by an edge are of different colors. If there is an edge between $y \in X_i$ and $z \in X_j$, where $i$ and $j$ are both either even or odd, then we have a cycle $x_0, y, z, \ldots, x_0$ of length $i + j + 1$, and this value is even, leading to a contradiction. The coloring, therefore, is valid, which concludes the proof.

In a bipartite graph, if we partition vertices into two sets according to the colors, all edges go from one set to the other. We retrieve here the usual definition of bipartite graphs, namely graphs whose vertices are partitioned into two sets and with no edge inside these sets. We now consider colorings of general graphs, and we propose a few greedy algorithms to solve the problem.

3.3.2 Greedy algorithms to color general graphs

The first greedy algorithm takes the vertices in a random order, and, for each vertex $v$, it colors it with the smallest color number that has not been yet given to a neighbor of $v$, i.e., a node connected to $v$.

Let $K_{\text{greedy}1}$ be the total number of colors needed by this greedy algorithm. Then we have $K_{\text{greedy}1} \leq \Delta(G) + 1$, where $\Delta(G)$ is the maximal degree of a vertex (number of edges of the vertex). Indeed, at any step of the algorithm, when we color vertex $v$, it has at most $\Delta(G)$ neighboring vertices and, therefore, the greedy algorithm never needs to use more than $\Delta(G) + 1$ colors.

Note that this algorithm is optimal for a fully connected graph (a clique), since we need $\Delta(G) + 1$ colors to connect such a graph (one color per vertex). However, this algorithm is not optimal in general; on the following bipartite graph, if the order of coloring is 1 and then 4, we need three colors, while the optimal coloring uses only two colors.

```
1 ————>
/    \\
3       4
```

In order to improve the previous algorithm, one idea consists in ordering vertices in a smart way, and then in proceeding as before, i.e., color each vertex in turn with the smallest possible color.

Let $n = |V|$ be the number of vertices, and $d_i$ be the degree of vertex $v_i$. We have $K_{\text{greedy}2} \leq \max_{1 \leq i \leq n} \min(d_i + 1, i)$. Indeed, when we color vertex $v_i$, it has at most $\min(d_i, i - 1)$ neighbors that have already been colored, and thus its own color is at most $1 + \min(d_i, i - 1) = \min(d_i + 1, i)$. To obtain the result, we take the maximum of these values on all vertices.

This result suggests that it would be smart to first color vertices with a high degree, so that we have $\min(d_i + 1, i) = i$. Therefore, the second greedy algorithm sorts the vertices by nonincreasing degrees.

Once again, the algorithm is not optimal. On the following bipartite graph, we choose to color vertex 1, then vertex 4, which imposes the use of three colors instead of the two required ones.

```
1 ————>
/    \\
3       4
```
Based on these ideas, several greedy algorithms can be designed. In particular, a rather intuitive idea consists in giving priority to coloring vertices that have already many colored neighbors. We define the color-degree of a vertex as the number of its neighbors that are already colored. Initially, the color-degree of each vertex is set to 0, and then it is updated at each step of the greedy algorithm.

The following greedy algorithm is called the Dsatur algorithm in [20]. The ordering is done by (color-degree, degree); we choose a vertex $v$ with maximum color-degree, and such that its degree is the largest among the vertices with maximum color-degree. This vertex $v$ is then colored with the smallest possible color, and the color-degrees of the neighbors of $v$ are updated before proceeding to the next step of the algorithm. We illustrate this algorithm on the following example:

We first choose a vertex with maximum degree, for instance $v_1$, and it is colored with color 1. The color-degree of $v_2$, $v_5$, $v_6$, and $v_7$ becomes 1, and we choose $v_2$, which has the maximum degree (between these four vertices); it is assigned color 2. Now, $v_7$ is the only vertex with color-degree 2; it is given the color 3. All remaining noncolored vertices have the same color-degree 1 and the same degree 3, we arbitrarily choose $v_3$ and color it with 1. Then, $v_4$, with color-degree 2, receives color 3. Finally, $v_5$ is colored in 2 and $v_6$ in 3; the graph is 3-colored, and it is an optimal coloring.

The name Dsatur comes from the fact that maximum color-degree vertices are saturated first. We prove below that Dsatur always returns an optimal coloring on bipartite graphs; however, it may use more colors than needed on arbitrary graphs.

**Theorem 3.2.** The Dsatur algorithm is optimal on bipartite graphs, i.e., it always succeeds to color them with two colors.

**Proof.** Consider a connected bipartite graph $G = (V,E)$, where $V = B \cup R$ and each edge in $E$ is connecting a vertex in $B$ (color 1 is blue) and a vertex in $R$ (color 2 is red). Note first that the first two greedy algorithms may fail. Let $G$ be such that $B = \{b_1, b_2, b_3\}$, $R = \{r_1, r_2, r_3\}$, and $E = \{(b_1, r_2), (b_2, r_3), (b_3, r_1), (b_1, r_1)\}$, as illustrated below.
3.3. Graph coloring

All vertices have a degree 2. If we start by coloring a vertex of \( B \), for instance \( b_1 \), and then a nonconnected vertex of \( R \), \( r_3 \), with the same color 1, it is not possible to complete the coloring with only two colors. The use of the color-degree prevents us from such a mistake, since once \( b_1 \) has been colored, we need to color either \( r_1 \) or \( r_2 \) with the color 2, and finish the coloring optimally.

In the general case, with \( Dsatur \), we first color a vertex, for instance from \( B \), with color 1 (blue). Then we have to color a vertex of color-degree 1, that is, a neighboring vertex. This neighboring vertex belongs necessarily to \( R \). It is colored with color 2 (red). We prove by induction that at any step of the algorithm, all colored vertices of \( B \) are colored in blue, and all vertices of \( R \) are colored in red. Indeed, if the coloring satisfies this property at a given step of the algorithm, we choose next a vertex \( v \) with nonnull color-degree. Because the graph is bipartite, all its neighbors are in the same set and have the same color: red if \( v \in B \), or blue if \( v \in R \). Vertex \( v \), therefore, is colored in red if it is in \( R \), or in blue if it is in \( B \).

We exhibit a counterexample to show that \( Dsatur \) is not optimal on arbitrary graphs.

\[ Dsatur \] can choose \( v_4 \) first, because it has the maximum degree 3; it is colored with 1. Between the vertices with color-degree 1, the algorithm can (arbitrarily) choose \( v_5 \), which is colored with 2. Then the algorithm can choose to color \( v_6 \), using color 1. Then, \( v_1 \) is chosen between vertices of color-degree 1 and degree 3, and it is colored with 2. We finally need to use colors 3 and 4 for \( v_3 \) and \( v_9 \), while this graph could have been colored with only three colors (\( v_1, v_5, v_7 \) with color 1, \( v_2, v_8, v_9 \) with color 2, and \( v_3, v_4 \) with color 3).

To build this counterexample, we force \( Dsatur \) to make a wrong decision, by coloring both \( v_4 \) and \( v_6 \) with color 1, and \( v_1 \) with color 2, which forces four colors because of \( v_2 \) and \( v_3 \). Note that it would be easy to build an example without any tie (thereby avoiding random choices) by increasing the degree of some vertices (for instance, in the example, \( v_7 \) and \( v_9 \) are just there to increase the degrees of \( v_1 \) and \( v_6 \)).

The problem of coloring general graphs is NP-complete, as will be shown in Chapter 7. However, for a particular class of graphs, a smart greedy algorithm can return the optimal solution, as we detail below.
Chapter 3. Greedy algorithms

3.3.3 Coloring interval graphs

We focus now on interval graphs. Given a set of intervals, we define a graph whose vertices are intervals, and whose edges connect intersecting intervals. The following example shows such a graph, obtained with a set of seven intervals.

```
  a———d———f——
  
  b——c——e——g——

  a———d———f——
  
  b——c——e——g——
```

The problem of coloring such a graph is quite similar to the sports hall problem. Indeed, one can see each interval as representing an event, with its starting and ending time, and the color as representing a sports hall. Then, only compatible events will be colored with the same color, and we could use one sports hall per set of compatible events. If we minimize the number of colors, we minimize the number of sports halls that are needed to organize all events.

Graphs that are obtained from a set of intervals are called interval graphs. We define the following greedy algorithm: intervals (i.e., vertices) are sorted by nondecreasing starting times (or left extremity). In the example, the order is \(a, b, c, d, e, f, g\). Then, the greedy coloring is done as before; for each chosen vertex, we color it with the smallest compatible color. On the example, we obtain the coloring \(1, 2, 3, 1, 1, 2, 3\), which is optimal, as the graph contains a cycle of length 3.

We prove now that this greedy algorithm is optimal for any interval graph. Let \(G\) be such a graph, and let \(d_v\) be the starting time of interval \(v\) corresponding to vertex \(v\). We execute the greedy algorithm; it uses \(k\) colors. If vertex \(v\) receives color \(k\), then this means that \(k - 1\) intervals that start no later than \(d_v\) intersect this interval and had all been colored with colors 1 to \(k - 1\); otherwise, \(v\) would be colored with a color \(c \leq k - 1\). All of these intervals are thus intersecting, because they all contain the point \(d_v\); therefore, graph \(G\) contains a clique of size \(k\). Since all vertices of a clique must be colored with distinct colors, we cannot color the graph with less than \(k\) colors. The greedy algorithm, therefore, is optimal.

Once again, we point out that the order chosen by the greedy algorithm is vital, since we could force the greedy algorithm to make a wrong decision, even on a bipartite graph as below, if we would not proceed from left to right. We could first color \(a\), then \(d\), leading to the use of three colors instead of two.
3.4 Theory of matroids

In this section are elementary results on matroids, a framework that allows us to guarantee the optimality of a generic greedy algorithm in some situations. Unfortunately, it is not easy to characterize which problems can be captured as matroid instances. Still, the theory is beautiful, and we outline its main ideas.

Matroids. The term matroid was introduced in 1935 by H. Whitney [108], while working on the linear independence of the vector columns of a matrix. We define it below and illustrate the concept through a canonical example.

**DEFINITION 3.1.** $(S, \mathcal{I})$ is a matroid if $S$ is a set of $n$ elements, and $\mathcal{I}$ is a collection of subsets of $S$, with the following properties:

i. $X \in \mathcal{I} \Rightarrow (\forall Y \subset X, Y \in \mathcal{I})$ (hereditary property), and

ii. $(A, B \in \mathcal{I}, |A| < |B|) \Rightarrow \exists x \in B \setminus A$ s.t. $A \cup \{x\} \in \mathcal{I}$ (exchange property).

If $X \in \mathcal{I}$, $X$ is said to be an independent set.

Readers familiar with linear algebra will immediately see that linearly independent subsets of a given vector set form a matroid. The canonical computer science example follows.

**Example of matroid: Forests of a graph.** Let $G = (V, E)$ be a (nondirected) graph. We define a matroid with $S = E$ (the elements are the edges of the graph), and $\mathcal{I} = \{A \subseteq E \mid A$ has no cycle$\}$. Therefore, a set of edges is an independent set if and only if this set of edges is a forest of the graph, i.e., a set of trees (a tree is a connected graph with no cycle). We check that this matroid satisfies both properties.

(i) The hereditary property. It is pretty obvious that a subset of a forest is a forest; if we remove edges from a forest, we cannot create a cycle, thus we still have a forest.

(ii) The exchange property. Let $A$ and $B$ be two forests of $G$ (i.e., $A, B \in \mathcal{I}$) such that $|A| < |B|$. $|A|$ is the number of edges in forest $A$, and every vertex is part of a tree (an isolated vertex with no edges is a tree made of a single vertex). Then $A$ (resp. $B$) contains $|V| - |A|$ (resp. $|V| - |B|$) trees. Indeed, each time an edge is added to the independent set, two trees are connected, therefore decrementing the number of trees by one. Thus, $B$ contains less trees than $A$, and there exists a tree $T$ of $B$ that is not included in a tree of $A$, i.e., two vertices $u$ and $v$ of tree $T$ are not in the same tree of $A$. On the path from $u$ to $v$ in $T$, there are two vertices, connected by an edge $(x, y)$, that are not in the same tree of $A$. Then, if we add this edge to the forest $A$, we still have a forest, i.e., $A \cup \{(x, y)\} \in \mathcal{I}$, which concludes the proof.
DEFINITION 3.2. Let \( F \in \mathcal{I} \); \( x \notin F \) is an extension of \( F \) if \( F \cup \{x\} \in \mathcal{I} \), i.e., \( F \cup \{x\} \) is an independent set. An independent set is maximal if it has no extensions.

In our running example, any edge connecting two distinct trees of a forest is an extension. A forest is maximal if adding any edge to it would create a cycle. A maximal independent set in the example of the forest is a spanning tree (or spanning forest if \( G \) is not connected).

LEMMA 3.1. All maximal independent sets are of same cardinal.

Proof. If this lemma was not true, we could find an extension to the independent set of smaller cardinal thanks to the exchange property, which would mean that it was not maximal.

We introduce a last definition: We add weights to the elements of the matroid and, therefore, obtain a weighted matroid.

DEFINITION 3.3. In a weighted matroid, each element of \( S \) has a weight: \( x \in S \mapsto w(x) \in \mathbb{N} \). The weight of a subset \( X \subseteq S \) is defined as the sum of the weights of its elements: \( w(X) = \sum_{x \in X} w(x) \).

Greedy algorithms on a weighted matroid. The problem is to find an independent set of maximum weight. The idea of the greedy algorithm is to sort elements of \( S \) by nonincreasing weights. We start with the empty set, which always is an independent set because of the hereditary property. Then we add elements into this set, as long as we keep an independent set. This generic algorithm is formalized in Algorithm 3.1.

ALGORITHM 3.1: Independent set of maximum weight.

1 Sort elements of \( S = \{s_1, \ldots, s_n\} \) by nonincreasing weight: \( w(s_1) \geq w(s_2) \geq \cdots \geq w(s_n) \)
2 \( A \leftarrow \emptyset \)
3 for \( i = 1 \) to \( n \) do
4 \hspace{1em} if \( A \cup \{s_i\} \in \mathcal{I} \) then
5 \hspace{1em} \hspace{1em} \( A \leftarrow A \cup \{s_i\} \)

THEOREM 3.3. Algorithm 3.1 returns an optimal solution to the problem of finding an independent set of maximum weight in the weighted matroid.

Proof. Let \( s_k \) be the first independent element of \( S \), i.e., the first index \( i \) of the algorithm such that \( \{s_i\} \subseteq \mathcal{I} \). We first prove that there exists an optimal solution that contains \( s_k \).
3.4. Theory of matroids

Let $B$ be an optimal solution, i.e., an independent set of maximum weight. If $s_k \in B$, we are done. Otherwise, let $A = \{s_k\} \in \mathcal{I}$. While $|B| > |A|$, we apply the exchange property to add an element of $B$ to the independent set $A$. We obtain the independent set with $|B|$ elements, $A = \{s_k\} \cup B \setminus \{s_j\}$, where $\{s_j\}$ is the one element of $B$ that has not been chosen for the extension (there is already element $s_k$ in $A$, and at the end, $|A| = |B|$, therefore all elements of $B$ but one are extensions of $A$).

We now compare the weights. We have $w(A) = w(B) - w(s_j) + w(s_k)$. Moreover, $w(s_k) \geq w(s_j)$, because $s_j$ is independent (by hereditary property), and $j > k$ (by definition of $s_k$). Finally, $w(A) \geq w(B)$, and since $B$ is an optimal solution, $w(A) = w(B)$. The independent set $A$ is of maximal weight, and it contains $s_k$, which proves the result.

To prove the theorem, we show by induction that the greedy algorithm returns the optimal solution; we restrict the search to a solution that contains $s_k$, and we start the reasoning again with $S' = S \setminus \{s_k\}$, and $\mathcal{I}' = \{X \subset S' \mid X \cup \{s_k\} \in \mathcal{I}\}$. 

Back to the running example. Theorem 3.3 proves the optimality of Kruskal’s algorithm to build a minimum weight spanning tree [69]. Edges are sorted by nondecreasing weight, and we choose greedily the next edge that does not add a cycle when added to the current set of edges. Of course, we should discuss a suitable data structure so that we can easily check the condition “no cycle has been created.” With a simple array, we can check the condition in $O(n^2)$, and it is possible to achieve a better complexity with other data structures [28]. In any case, the complexity of the greedy algorithm remains polynomial.

Example: A semimatching problem. In this very simple example, we are given a directed weighted graph. The problem is to find a maximum weight subset of the edges so that no two starting points are the same. A natural greedy algorithm would sort all edges according to their weight in nonincreasing order, then consider all edges in this order, selecting an edge $(i,j)$ if and only if no edge $(i,j')$ had been selected earlier. In fact, this greedy algorithm selects for every node the outgoing edge that has maximum weight, hence, it can be easily implemented in time $O(n+m)$, where $n$ is the number of nodes, and $m$ the number of edges, of the directed graph. While the optimality of this greedy algorithm is not difficult to prove directly, we prove it using matroid theory.

The problem can be cast in terms of a matrix $W$ with nonnegative entries, with the goal to select a set of entries whose sum is maximal, subject to the constraint that no two entries are from the same row of the matrix. There are $n$ rows in $W$, one per node in the graph. Let $W_{ij}$ be the entry in row $i$ and column $j$ of the matrix $W$, and let $x_{ij} \in \{0,1\}$ be the indicator of
whether \( W_{ij} \) is selected. We aim at maximizing \( \sum_{i,j} W_{ij}x_{ij} \) subject to the set of constraints \( \sum_j x_{ij} \leq 1 \) for each row \( i \). The greedy algorithm chooses entries one at a time in order of weight, largest first (and breaking ties arbitrarily), rejecting an entry only if an entry in the same row has already been chosen. Here is an example, where chosen entries are underlined:

\[
W = \begin{pmatrix}
12 & 7 & 10 & 11 \\
8 & 6 & 4 & 16 \\
3 & 5 & 2 & 1 \\
14 & 13 & 9 & 15
\end{pmatrix}
\]

To prove that the greedy algorithm is optimal, we exhibit the matroid; independent sets are sets of entries such that no two of them are from the same row of the matrix. We show that both properties hold. The hereditary property is obvious. Indeed, when removing entries from an independent set, we cannot create a row with two entries or more. The exchange property is not difficult either. Let \( A \) and \( B \) be two independent sets with \( |A| < |B| \). There is at most one element per row in \( A \) and \( B \), so there must be a row that contains an element of \( B \) and no element of \( A \). Adding this element to \( A \) preserves its independence. This concludes the proof of optimality of the greedy algorithm.

As mentioned before, it is not easy to exhibit matroid structures for which interesting and efficient greedy algorithms can be derived. A more complicated example that involves scheduling tasks with deadlines is studied in Exercise 3.5. We refer the reader to [72, 95] for much more material on matroids and greedoids.

### 3.5 Exercises

**Exercise 3.1: Interval cover** (solution p. 68)

We are given a set \( X = \{x_1, \ldots, x_n\} \) of \( n \) points on a line.

1. Design a greedy algorithm that determines the smallest set of closed intervals of length 1 that contains all the points.
2. Prove the optimality of the algorithm and give its complexity.
3. Could you use the theory of matroids to prove the optimality of the algorithm?

**Exercise 3.2: Memory usage** (solution p. 69)

Given a memory of size \( L \), we want to store a set of \( n \) files \( P = (P_1, \ldots, P_n) \). File \( P_i \) (\( 1 \leq i \leq n \)) is of size \( a_i \), where \( a_i \) is an integer. If \( \sum_{i=1}^{n} a_i > L \), we
cannot store all files. We need to select a subset $Q \subseteq P$ of files to store, such that $\sum_{P_i \in Q} a_i \leq L$. We sort the files $P_i$ by nondecreasing sizes ($a_1 \leq \cdots \leq a_n$).

1. Write a greedy algorithm that maximizes the number of files in $Q$. The output must be a Boolean table $S$ such that $S[i] = 1$ if $P_i \in Q$, and $S[i] = 0$ otherwise. What is the complexity of this algorithm, in number of comparisons and number of arithmetic operations?

2. Prove that this strategy always returns a maximal subset $Q$. We define the utilization ratio as $\frac{\sum_{P_i \in Q} a_i}{L}$. How small can it be with our strategy?

3. We now want to maximize the utilization ratio, i.e., fill the memory as much as possible. Design a greedy algorithm for this new objective function.

4. Is the latter greedy algorithm optimal? How small can the utilization ratio be with this algorithm? Prove the result.

**Exercise 3.3: Scheduling dependent tasks on several machines**

(solution p. 71)

Let $G = (V, E)$ be a directed acyclic graph (DAG). Here $G$ is a task graph. In other words, each node $v \in V$ represents a task, and each edge $e \in E$ represents a precedence constraint, i.e., if $e = (v_1, v_2) \in E$, then the execution of $v_2$ cannot start before the end of the execution of $v_1$. We need to schedule the tasks on an unlimited number of processors. Moreover, the execution time of task $v \in V$ is $w(v)$. The problem is to find a valid schedule, i.e., a start time $\sigma(v)$ for each task $v$ such that no precedence constraints are violated, and that minimizes the total execution time. The reader may refer to Section 6.4.4, p. 140, for more background on scheduling.

1. Define formally (by induction) the top level $tl(v)$ of a task $v \in V$, which is the earliest possible starting time of task $v$.

2. Propose a greedy schedule of the tasks, based on the top levels, and prove its optimality. This schedule is called $\sigma_{free}$.

3. We define the bottom level $bl(v)$ of a task as the largest weight of a path from $v$ to an output task, i.e., a task with no successor. The weight of the path includes the weight of $v$. Define bottom levels formally, and propose a schedule of the tasks, based on the bottom levels, that is called $\sigma_{late}$.

4. Show that any optimal schedule $\sigma$ satisfies:

$$\forall v \in V, \ \sigma_{free}(v) \leq \sigma(v) \leq \sigma_{late}(v).$$
5. Give an example of a DAG that has at least three different optimal schedules.

Exercise 3.4: Scheduling independent tasks with priorities
(solution p. 72)

We need to schedule $n$ independent tasks, $T_1, T_2, \ldots, T_n$, on a single processor. Each task $T_i$ has an execution time $w_i$ and a priority $p_i$. Because we execute the tasks sequentially on a single processor and as we target an optimal schedule, we can focus on schedules that execute tasks as soon as possible. A schedule is then fully defined by the order followed to execute the tasks. In other words, here, a schedule of tasks $T_1, T_2, \ldots, T_n$ is a permutation $T_{\sigma(1)}, T_{\sigma(2)}, \ldots, T_{\sigma(n)}$, specifying the order in which tasks are executed. We assume that the first task to be executed is processed from time 0 on. The cost of a schedule is defined as $\sum_{i=1}^{n} p_i C_i$, where $C_i$ is the completion time of task $T_i$, i.e., the date at which its processing was completed. We look for a schedule that minimizes this cost.

1. Consider any schedule and two tasks $T_i$ and $T_j$ that are executed consecutively under this schedule. Which task should be executed first in order to minimize the cost?

2. Design an optimal greedy algorithm. What is its complexity?

Exercise 3.5: Scheduling independent tasks with deadlines
(solution p. 73)

The goal here is to exhibit a matroid to prove the optimality of a greedy algorithm. We need to schedule $n$ independent tasks, $T_1, T_2, \ldots, T_n$, on a single processor. Each task $T_i$ is executed in one time unit, but it has a deadline $d_i$ that should not be exceeded. If a task does not complete its execution before its deadline, there is a cost $w_i$ to pay. The objective here is to find a schedule that minimizes the sum of the costs of the tasks that are completed after their deadlines. A schedule, in this exercise, will be a function, $\sigma : T \rightarrow \mathbb{N}$, that associates to each task its execution time, such that two tasks cannot be scheduled at the same time, i.e., for all $1 \leq i, j \leq n$, $\sigma(T_i) \neq \sigma(T_j)$. The first task can be executed at time 0.

We say that a task is on time if it finishes its execution before its deadline, and that it is late otherwise. Note that minimizing the cost of late tasks is equivalent to maximizing the cost of on-time tasks. A canonical schedule is such that (i) on-time tasks are scheduled before late tasks, and (ii) on-time tasks are ordered by nondecreasing deadlines.

1. Prove that there is always an optimal schedule that is canonical, i.e., we can restrict the search to canonical schedules.
2. Design a greedy scheduling algorithm to solve the problem. What is its complexity?

3. Illustrate the greedy algorithm on the following example with seven tasks; the tasks are sorted by nonincreasing weight $w_i$, for $1 \leq i \leq 7$, and their deadlines are as follows: $d_1 = 4, d_2 = 2, d_3 = 4, d_4 = 3, d_5 = 1, d_6 = 4,$ and $d_7 = 6$.

4. Prove the optimality of the algorithm by exhibiting a matroid.

While Exercises 3.4 and 3.5 deal with simple uniprocessor scheduling problems for which the greedy algorithm is optimal, there are many more complex scheduling problems [21]. These include, for instance, scheduling problems with tasks with different execution times, several machines, precedence constraints between tasks, and so on. More scheduling problems are described in Section 6.4.4, p. 140.

**Exercise 3.6: Edge matroids** (solution p. 74)

This exercise aims at illustrating the matroid theory. The goal here is to exhibit a weighted matroid, to design the corresponding greedy algorithm, and prove its optimality.

This exercise is a generalization of the semimatching algorithm presented in Section 3.4. We are given a directed graph $G = (V,E)$ whose edges have integer weights. Let $w(e)$ be the weight of edge $e \in E$. We also are given a constraint $f(u) \geq 0$ on the out-degree of each node $u \in V$. The goal is to find a subset of edges of maximal weight, and whose out-degree at any node satisfy the constraint. We see that if $f(u) = 1$ for all nodes, we retrieve the semimatching algorithm.

1. Define independent sets and prove you have a matroid.

2. What is the cardinal of maximal independent sets?

3. What is the complexity of the (optimal) greedy algorithm?

**Exercise 3.7: Huffman code** (solution p. 75)

Let $\Sigma$ be a finite alphabet with at least two elements. A binary code is an injective application from $\Sigma$ to the set of finite suites of 0 and 1 (i.e., a binary word, also called code word). The code can be naturally extended by concatenation to a mapping defined on the set $\Sigma^*$ of words using the alphabet $\Sigma$. A code is said to be of fixed length if all the letters in $\Sigma$ are coded by binary words of same size. A code is said to be prefix if no code word is a prefix of another code word. Given the code of a word in $\Sigma^*$, the decoding operation consists in finding the original word.
1. Prove that the decoding operation has a unique solution, both for a code of fixed length, and for a prefix code.

2. Represent a prefix code by a binary tree, where leaves are the letters of the alphabet $\Sigma$.

3. Consider a text in which each letter $c \in \Sigma$ appears with a frequency $f(c) \neq 0$. To each prefix code of this text, represented by a tree $T$, is associated a cost, defined by $B(T) = \sum_{c \in \Sigma} f(c) \times l_T(c)$, where $l_T(c)$ is the size of the code word of $c$. If $f(c)$ is exactly the number of occurrences of $c$ in the text, then $B(T)$ is the number of bits in the encoded text. A prefix code $T$ is optimal if, for this text, $B(T)$ is minimum. Prove that for any optimal prefix code there is a corresponding binary tree with $|\Sigma|$ leaves and $|\Sigma| - 1$ internal nodes.

4. Prove that there is an optimal prefix code such that two letters of smallest frequencies are siblings in the tree (i.e., their code words have the same size, and differ only by the last bit).

   Hint: Prove also that these two letters are leaves of maximal depths.

5. Given $x$ and $y$, two letters of smallest frequencies, we consider the alphabet $\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$, where $z$ is a new letter with frequency $f(z) = f(x) + f(y)$. Let $T'$ be the tree of an optimal code for $\Sigma'$. Prove that the tree $T$ obtained from $T'$ by replacing the leave associated to $z$ by an internal node with two leaves $x$ and $y$ is an optimal code for $\Sigma$.

6. Using both previous questions, design an algorithm that returns an optimal code, and give its complexity. Illustrate the algorithm on the following problem instance: $\Sigma = \{a, b, c, d, e, g\}$, $f(a) = 45$, $f(b) = 13$, $f(c) = 12$, $f(d) = 16$, $f(e) = 9$, and $f(g) = 5$.

### 3.6 Solutions to exercises

**Solution to Exercise 3.1: Interval cover**

1. Algorithm 3.2 is a greedy algorithm to solve the interval cover problem. It builds a maximal length interval starting at the first point on the line, remove all points included in that interval, and then iterates.

2. Sorting the points cost $O(n \log(n))$ when the execution of the while loop costs $O(n)$. Therefore, the algorithm runs in $O(n \log(n))$.

   We prove the optimality of Algorithm 3.2 by induction on the number $n$ of points. If $n = 1$, there is a single point, the algorithm returns a single interval and thus is optimal. Now, assume we have proved the